CS 277: Database System Implementation

Notes 10: More TP

Arthur Keller
Sections to Skim:

• Chapter 17: none (read all sections)
• Chapter 18:
  – skim 18.8
• Chapter 19:
  – skim 19.4, 19.5, 19.6, 19.7
  – maybe 19.2 (decide later...)
• Chapter 20: none (read all sections)
Chapter 19  More on transaction processing

Topics:
- Cascading rollback, recoverable schedule
- Deadlocks
  - Prevention
  - Detection
- View serializability
- Distributed transactions
- Long transactions (nested, compensation)
Concurrency control & recovery

Example:

\[
\begin{array}{ll}
T_j & Ti \\
\vdots & \vdots \\
W_j(A) & \vdots \\
\vdots & ri(A) \\
\vdots & \text{Commit } Ti \\
\vdots & \\
\text{Abort } T_j & \vdots \\
\end{array}
\]

Cascading rollback (Bad!)
• Schedule is conflict serializable
• $T_j \rightarrow T_i$

• But not recoverable
• Need to make “final” decision for each transaction:
  – **commit decision** - system guarantees transaction will or has completed, no matter what
  – **abort decision** - system guarantees transaction will or has been rolled back (has no effect)
To model this, two new actions:

- $C_i$ - transaction $T_i$ commits
- $A_i$ - transaction $T_i$ aborts
Back to example:

\[
\begin{align*}
T_j & \quad T_i \\
\vdots & \quad \vdots \\
W_j(A) & \quad r_i(A) \\
\vdots & \\
C_i & \leftarrow \text{can we commit here?}
\end{align*}
\]
Definition

$T_i$ reads from $T_j$ in $S$ ($T_j \Rightarrow_S T_i$) if

(1) $w_j(A) <_S r_i(A)$

(2) $a_j \prec_S r_i(A)$ \hspace{1cm} (\prec: does not precede)

(3) If $w_j(A) <_S w_k(A) <_S r_i(A)$ then $a_k <_S r_i(A)$
Definition

Schedule $S$ is recoverable if whenever $T_j \Rightarrow_S T_i$ and $j \neq i$ and $C_i \in S$ then $C_j <_S C_i$
Note: in transactions, reads and writes precede commit or abort

If \( C_i \in T_i \), then \( r_i(A) < C_i \)
\[ w_i(A) < C_i \]

If \( A_i \in T_i \), then \( r_i(A) < A_i \)
\[ w_i(A) < A_i \]

• Also, one of \( C_i, A_i \) per transaction
How to achieve recoverable schedules?
With 2PL, hold write locks to commit (strict 2PL)

\[
\begin{array}{c|c}
T_j & T_i \\
\vdots & \vdots \\
W_j(A) & \vdots \\
\vdots & \vdots \\
C_j & \vdots \\
u_j(A) & \vdots \\
x_j(A) & r_i(A)
\end{array}
\]
With validation, no change!
• S is recoverable if each transaction commits only after all transactions from which it read have committed.

• S avoids cascading rollback if each transaction may read only those values written by committed transactions.
• S is strict if each transaction may *read and write* only items previously written by committed transactions.
Where are serializable schedules?

- RC
- ACR
- ST
- SERIAL
Examples

• Recoverable:
  \[ w_1(A) \; w_1(B) \; w_2(A) \; r_2(B) \; c_1 \; c_2 \]

• Avoids Cascading Rollback:
  \[ w_1(A) \; w_1(B) \; w_2(A) \; c_1 \; r_2(B) \; c_2 \]

• Strict:
  \[ w_1(A) \; w_1(B) \; c_1 \; w_2(A) \; r_2(B) \; c_2 \]

Assumes \( w_2(A) \) is done without reading
Deadlocks

• Detection
  – Wait-for graph

• Prevention
  – Resource ordering
  – Timeout
  – Wait-die
  – Wound-wait
Deadlock Detection

• Build Wait-For graph
• Use lock table structures
• Build incrementally or periodically
• When cycle found, rollback victim
Resource Ordering

- Order all elements $A_1, A_2, \ldots, A_n$
- A transaction $T$ can lock $A_i$ after $A_j$ only if $i > j$

Problem: Ordered lock requests not realistic in most cases
Timeout

• If transaction waits more than L sec., roll it back!
• Simple scheme
• Hard to select L
Wait-die

- Transactions given a timestamp when they arrive .... ts(T_i)
- T_i can only wait for T_j if ts(T_i) < ts(T_j)  
  ...else die
Example:

T₁
(ts = 10)

wait?

wait

T₂
(ts = 20)

T₃
(ts = 25)
Second Example:

T1 (ts =22)

requests A: wait for T2 or T3?

T2 (ts =20)

wait(A)

T3 (ts =25)

Note: ts between 20 and 25.
Second Example (continued):

One option: $T_1$ waits just for $T_3$, transaction holding lock. But when $T_2$ gets lock, $T_1$ will have to die!

\[
\begin{align*}
T_1 & \quad \text{(ts = 22)} \\
& \quad \text{wait}(A) \\
T_3 & \quad \text{(ts = 25)} \\
T_2 & \quad \text{(ts = 20)} \\
& \quad \text{wait}(A)
\end{align*}
\]
Second Example (continued):

Another option: $T_1$ only gets A lock after $T_2$, $T_3$ complete, so $T_1$ waits for both $T_2$, $T_3$ $\Rightarrow$ $T_1$ dies right away!
Second Example (continued):

Yet another option: $T_1$ preempts $T_2$, so $T_1$ only waits for $T_3$; $T_2$ then waits for $T_3$ and $T_1$... $\implies$ $T_2$ may starve?

- $T_1$ $(ts = 22)$
- $T_2$ $(ts = 20)$
- $T_3$ $(ts = 25)$

$T_2$ may starve? (冗余弧)
Wound-wait

- Transactions given a timestamp when they arrive ... $ts(T_i)$
- $T_i$ wounds $T_j$ if $ts(T_i) < ts(T_j)$
  else $T_i$ waits

“Wound”: $T_j$ rolls back and gives lock to $T_i$
Example:

T1
(ts = 25)

T3
(ts = 10)

wait

T2
(ts = 20)

wait

wait
Second Example:

\begin{itemize}
  \item \textbf{T}_1
    \begin{itemize}
      \item (ts = 15)
      \item \text{requests A: wait for T}_2 \text{ or T}_3?\end{itemize}
  \item \textbf{T}_2
    \begin{itemize}
      \item (ts = 20)
      \item \text{wait(A)}\end{itemize}
  \item \textbf{T}_3
    \begin{itemize}
      \item (ts = 10)
    \end{itemize}
\end{itemize}

Note: ts between 10 and 20.
Second Example (continued):

One option: $T_1$ waits just for $T_3$, transaction holding lock. But when $T_2$ gets lock, $T_1$ waits for $T_2$ and wounds $T_2$.
Second Example (continued):

Another option: $T_1$ only gets A lock after $T_2$, $T_3$ complete, so $T_1$ waits for both $T_2$, $T_3$ $\Rightarrow$ $T_2$ wounded right away!

$T_1$
\[(ts = 15)\]

$T_2$
\[(ts = 20)\]

$T_3$
\[(ts = 10)\]
Second Example (continued):

Yet another option: T₁ preempts T₂, so T₁ only waits for T₃; T₂ then waits for T₃ and T₁... \( \Rightarrow \) T₂ is spared!

\[
\begin{align*}
&\text{T}_1 \\
&(ts = 15) \\
&\text{wait(A)} \\
&\text{wait(A)} \\
&\text{T}_2 \\
&(ts = 20) \\
&\text{wait(A)} \\
&\text{wait(A)} \\
&\text{T}_3 \\
&(ts = 10)
\end{align*}
\]
User/Program commands

Lots of variations, but in general
- Begin_work
- Commit_work
- Abort_work
Nested transactions

User program:

  ::

  Begin_work;

  ::

  ::

  ::

    If results_ok, then commit work
    else abort_work
Nested transactions

User program:

Begin_work;
    Begin_work;
        Begin_work;
            If results_ok, then commit work
            else {abort_work; try something else...}
        ;
    ;
    If results_ok, then commit work
    else abort_work
Parallel Nested Transactions

\[ T_1: \begin{align*}
&\text{begin-work} \\
&\text{parallel:} \\
&T_{11}: \begin{align*}
&\text{begin}\_\text{work} \\
&\text{commit}\_\text{work}
\end{align*} \\
&T_{12}: \begin{align*}
&\text{begin}\_\text{work} \\
&\text{commit}\_\text{work} \\
&\text{commit}\_\text{work}
\end{align*}
\]
Locking

What are we really locking?
Example:

\[
T_i : \\
\quad \text{Read record } r_1 \\
\quad : \\
\quad \text{Read record } r_1 \quad \text{do record locking} \\
\quad : \\
\quad \text{Modify record } r_3 \\
\quad : \\
\]
But underneath:

If we lock all data involved in read of R1, we may prevent an update to R2 (which may require reorganization within block)
Solution: view DB at two levels

Top level: record actions
  record locks
  undo/redo actions — logical

  e.g., Insert record(X,Y,Z)
  Redo: insert(X,Y,Z)
  Undo: delete
Low level: deal with physical details
latch page during action
(release at end of action)
Note: undo does not return physical DB to original state; only same logical state

e.g., Insert R3  Undo (delete R3)
Logging Logical Actions

• Logical action typically span one block (physiological actions)

• Undo/redo log entry specifies undo/redo logical action

• Challenge: making actions idempotent
  • Example (bad): redo insert ⇒ key inserted multiple times!
Solution: Add Log Sequence Number

Log record:
• LSN=26
• OP=insert(5,v2) into P
• ...

| sem | lsn=25 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3, v1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5, v2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| sem | lsn=26 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3, v1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5, v2</td>
</tr>
</tbody>
</table>
Still Have a Problem!

Make log entry for undo

lsn=27

lsn=24
3, v1
4, v2

lsn=25
3, v1

lsn=26
3, v1
5, v3

lsn=??
3, v1
5, v3
4, v2

T1
Del 4

T2
Ins 5

undo
Del 4
Compensation Log Records

- Log record to indicate undo (not redo) action performed

- Note: Compensation may not return page to exactly the initial state
## At Recovery: Example

**Log:**

<table>
<thead>
<tr>
<th>...</th>
<th>lsn=21</th>
<th>...</th>
<th>lsn=27</th>
<th>...</th>
<th>lsn=35</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td></td>
<td>T1</td>
<td></td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a1</td>
<td></td>
<td>a2</td>
<td></td>
<td>a2⁻¹</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p1</td>
<td></td>
<td>p2</td>
<td></td>
<td>p2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What to do with p2 (during T1 rollback)?

- If lsn(p2) < 27 then ... ?
- If 27 ≤ lsn(p2) < 35 then ... ?
- If lsn(p2) ≥ 35 then ... ?

Note: lsn(p2) is lsn of p copy on disk
Recovery Strategy

[1] Reconstruct state at time of crash
   – Find latest valid checkpoint, $Ck$, and let $ac$ be its set of active transactions
   – Scan log from $Ck$ to end:
     • For each log entry $[\text{lsn, page}]$ do: if $\text{lsn}(\text{page}) < \text{lsn}$ then redo action
     • If log entry is start or commit, update $ac$
Recovery Strategy

[2] Abort uncommitted transactions

– Set ac contains transactions to abort
– Scan log from end to $Ck$:
  • For each log entry (not undo) of an ac transaction, undo action (making log entry)
– For ac transactions not fully aborted, read their log entries older than $Ck$ and undo their actions
Example: What To Do After Crash

Log:

<table>
<thead>
<tr>
<th>chkpt</th>
<th>lsn=21 T1 a1 p1</th>
<th>lsn=27 T1 a2 p2</th>
<th>lsn=29 T1 a3 p3</th>
<th>lsn=31 T1 a3⁻¹ p3</th>
<th>lsn=35 T1 a2⁻¹ p2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
During Undo: Skip Undo's

Log:

chk pt ...
lsn=21 T1 a1 p1 ...
lsn=27 T1 a2 p2 ...
lsn=29 T1 a3 p3 ...
lsn=31 T1 a3⁻¹ p3 ...
lsn=35 T1 a2⁻¹ p2 ...

pointer to forward action
pointer to previous T1 action
Related idea: Sagas

• Long running activity: \( T_1, T_2, \ldots, T_n \)
• Each step/transaction Ti has a compensating transaction Ti-1
• Semantic atomicity: execute one of
  \[ \begin{align*}
  &- T_1, T_2, \ldots, T_n \\
  &- T_1, T_2, \ldots, T_{n-1}, T^{-1}_{n-1}, T^{-1}_{n-2}, \ldots, T^{-1}_1 \\
  &- T_1, T_2, \ldots, T_{n-2}, T^{-1}_{n-2}, T^{-1}_{n-3}, \ldots, T^{-1}_1 \\
  &\vdots \\
  &- T_1, T^{-1}_1 \\
  &- nothing
  \end{align*} \]
Summary

- Cascading rollback
  Recoverable schedule
- Deadlock
  - Prevention
  - Detection
- Nested transactions
- Multi-level view